

05 | 面向对象编程：怎样才能写出一个“好”的类？

2020-05-16 罗剑锋

罗剑锋的C++实战笔记

[进入课程 >](#)



讲述：Chrono

时长 13:10 大小 12.08M



你好，我是 Chrono。

如果按照前几节课的惯例，今天应该是讲运行阶段的。但是，运行阶段跟前面的编码、预处理和编译阶段不同，它是动态的、实时的，内外部环境非常复杂，CPU、内存、磁盘、信号、网络套接字.....各种资源交织在一起，可谓千变万化（正如我在[第 1 节课](#)里所说，每一个阶段的差异都非常大）。

解决这个阶段面临的问题已经不是编程技术了，更多的是要依靠各种调试、分析、日志工具，比如 GDB、Valgrind、Systemtap 等。



所以，我觉得把这些运行阶段的工具、技巧放在课程前面不是太合适，咱们还是往后延一延，等把 C++ 的核心知识点都学完了，再来看它比较好。

那么，今天要和你聊哪些内容呢？

我想了想，还是讲讲“面向对象编程”（Object Oriented Programming）吧。毕竟，它是 C++ 诞生之初“安身立命”的看家本领，也是 C++ 的核心编程范式。

不管我们是否喜欢，“面向对象”早就已经成为了编程界的共识和主流。C++、Java、Python 等流行的语言，无一不支持面向对象编程，而像 Pascal、BASIC、PHP 那样早期面向过程的语言，在发展过程中也都增加了对它的支持，新出的 Go、Swift、Rust 就更不用说了。

毫无疑问，**掌握“面向对象编程”是现在程序员的基本素养**。但落到实际开发时，每个人对它的理解程度却有深有浅，应用的水平也有高有低，有的人设计出的类精致灵活，而有的人设计出来的却是粗糙笨重。

细想起来，“面向对象”里面可以研究的地方实在是太多了。那么，到底“面向对象”的精髓是什么？怎样才能用好它？怎样才能写出一个称得上是“好”的类呢？

所以，今天我就从设计思想、实现原则和编码准则这几个角度谈谈我对它的体会心得，以及在 C++ 里应用的一些经验技巧，帮你写出更高效、更安全、更灵活的类。（在第 19、20 课，我还会具体讲解，到时候你可以参考下。）

设计思想

首先要说的是，虽然很多语言都内建语法支持面向对象编程，但它本质上是一种设计思想、方法，与语言细节无关，要点是**抽象**（Abstraction）和**封装**（Encapsulation）。

掌握了这种代码之外的思考方式，就可以“高屋建瓴”，站在更高的维度上去设计程序，不会被语言、语法所限制。

所以，即使是像 C 这样“纯”面向过程的编程语言，也能够应用面向对象的思想，以 struct 实现抽象和封装，得到良好的程序结构。

面向对象编程的基本出发点是“对现实世界的模拟”，把问题中的实体抽象出来，封装为程序里的类和对象，这样就在计算机里为现实问题建立了一个“虚拟模型”。

然后以这个模型为基础不断演化，继续抽象对象之间的关系和通信，再用更多的对象去描述、模拟.....直到最后，就形成了一个由许多互相联系的对象构成的系统。

把这个系统设计出来、用代码实现出来，就是“面向对象编程”了。

不过，因为现实世界非常复杂，“面向对象编程”作为一种工程方法，是不可能完美模拟的，纯粹的面向对象也有一些缺陷，其中最明显的就是“继承”。

“继承”的本意是重用代码，表述类型的从属关系 (Is-A)，但它却不能与现实完全对应，所以用起来就会出现很多意外情况。

比如那个著名的长方形的例子。Rectangle 表示长方形，Square 继承 Rectangle，表示正方形。现在问题就来了，这个关系在数学中是正确的，但表示为代码却不太正确。长方形可以用成员函数单独变更长宽，但正方形却不行，长和宽必须同时变更。

还有那个同样著名的鸟类的例子。基类 Bird 有个 Fly 方法，所有的鸟类都应该继承它。但企鹅、鸵鸟这样的鸟类却不会飞，实现它们就必须改写 Fly 方法。

各种编程语言为此都加上了一些“补丁”，像 C++ 就有“多态”“虚函数”“重载”，虽然解决了“继承”的问题，但也使代码复杂化了，一定程度上扭曲了“面向对象”的本意。

实现原则

说了些“高大上”的理论，是不是有点犯迷糊？没关系，下面，我就在 C++ 里细化一下。

就像我刚才说的，“面向对象编程”的关键点是“抽象”和“封装”，而“继承”“多态”并不是核心，只能算是附加品。

所以，我建议你设计类的时候**尽量少用继承和虚函数**。

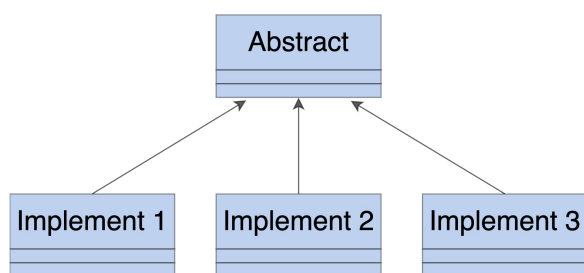
特别的，如果完全没有继承关系，就可以让对象不必承受“父辈的重担”（父类成员、虚表等额外开销），轻装前行，更小更快。没有隐含的重用代码也会降低耦合度，让类更独立，

更容易理解。

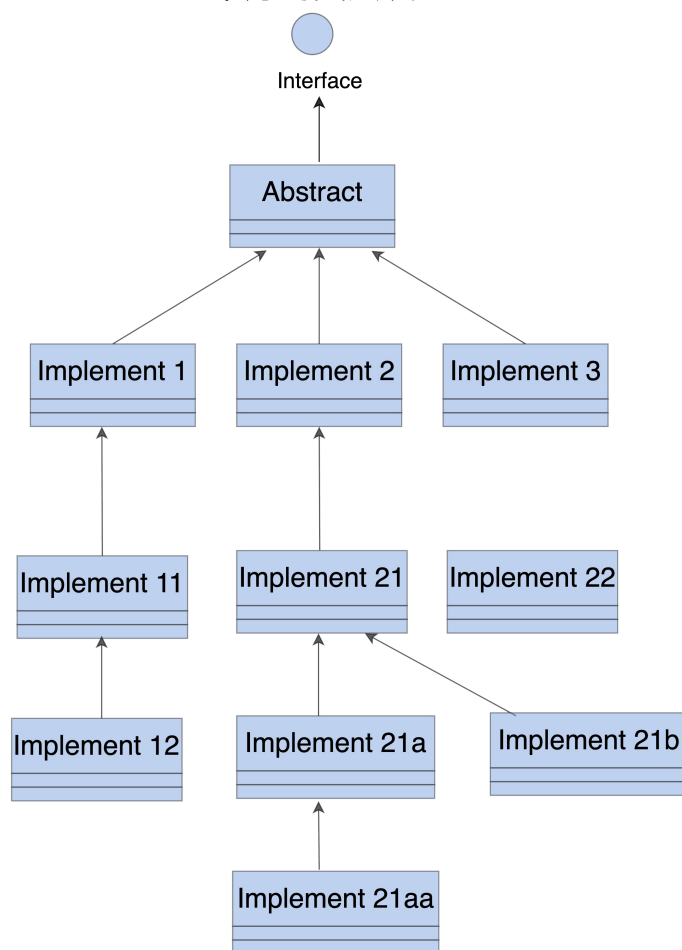
还有，把“继承”切割出去之后，可以避免去记忆、实施那一堆难懂的相关规则，比如 public/protected/private 继承方式的区别、多重继承、纯虚接口类、虚析构函数，还可以绕过动态转型、对象切片、函数重载等很多危险的陷阱，减少冗余代码，提高代码的健壮性。

如果非要用继承不可，那么我觉得一定要**控制继承的层次**，用 UML 画个类体系的示意图来辅助检查。如果继承深度超过三层，就说明有点“过度设计”了，需要考虑用组合关系替代继承关系，或者改用模板和泛型。

好的类层次



不好的类层次



在设计类接口的时候，我们也要让类尽量简单、“短小精悍”，**只负责单一的功能**。

如果很多功能混在了一起，出现了“万能类”“意大利面条类”（有时候也叫 God Class），就要应用设计模式、重构等知识，把大类拆分成多个各负其责的小类。

我还看到过很多人有一种不好的习惯，就是喜欢在类内部定义一些嵌套类，美其名曰“高内聚”。但恰恰相反，这些内部类反而与上级类形成了强耦合关系，也是另一种形式的“万能类”。

其实，这本来是名字空间该做的事情，用类来实现就有点“越权”了。正确的做法应该是，**定义一个新的名字空间，把内部类都“提”到外面，降低原来类的耦合度和复杂度。**

编码准则

有了这些实现原则，下面我再来讲几个编码时的细节，从安全和性能方面帮你改善类的代码。

C++11 新增了一个特殊的标识符“**final**”（注意，它不是关键字），把它用于类定义，就可以显式地禁用继承，防止其他人有意或者无意地产生派生类。无论是对人还是对编译器，效果都非常好，我建议你一定要积极使用。

 复制代码

```
1 class DemoClass final    // 禁止任何人继承我
2 { ... };
```

在必须使用继承的场合，建议你**只使用 public 继承，避免使用 virtual、protected**，因为它们会让父类与子类的关系变得难以捉摸，带来很多麻烦。当到达继承体系底层时，也要及时使用“final”，终止继承关系。

 复制代码

```
1 class Interface          // 接口类定义，没有final，可以被继承
2 { ... };
3
4 class Implement final : // 实现类，final禁止再被继承
5     public Interface    // 只用public继承
6 { ... };
```

C++ 里类的四大函数你一定知道吧，它们是构造函数、析构函数、拷贝构造函数、拷贝赋值函数。C++11 因为引入了右值（Rvalue）和转移（Move），又多出了两大函数：**转移构造函数**和**转移赋值函数**。所以，在现代 C++ 里，一个类总是会有六大基本函数：**三个构造、两个赋值、一个析构**。

好在 C++ 编译器会自动为我们生成这些函数的默认实现，省去我们重复编写的时间和精力。但我建议，对于比较重要的构造函数和析构函数，应该用 “= default” 的形式，明确地告诉编译器（和代码阅读者）：“应该实现这个函数，但我不想自己写。” 这样编译器就得到了明确的指示，可以做更好的优化。

 复制代码

```
1 class DemoClass final
2 {
3 public:
4     DemoClass() = default; // 明确告诉编译器，使用默认实现
5     ~DemoClass() = default; // 明确告诉编译器，使用默认实现
6 };
```

这种 “= default” 是 C++11 新增的专门用于六大基本函数的用法，相似的，还有一种 “= delete” 的形式。它表示**明确地禁用某个函数形式**，而且不限于构造 / 析构，可以用于任何函数（成员函数、自由函数）。

比如说，如果你想要禁止对象拷贝，就可以用这种语法显式地把拷贝构造和拷贝赋值 “delete” 掉，让外界无法调用。

 复制代码

```
1 class DemoClass final
2 {
3 public:
4     DemoClass(const DemoClass&) = delete; // 禁止拷贝构造
5     DemoClass& operator=(const DemoClass&) = delete; // 禁止拷贝赋值
6 };
```

因为 C++ 有隐式构造和隐式转型的规则，如果你的类里有单参数的构造函数，或者是转型操作符函数，为了防止意外的类型转换，保证安全，就要使用 “explicit” 将这些函数标记为 “显式”。

 复制代码

```
1 class DemoClass final
2 {
3 public:
4     explicit DemoClass(const string_type& str) // 显式单参构造函数
5     { ... }
6 }
```

```
7     explicit operator bool()                // 显式转型为bool
8     { ... }
9 };
```

常用技巧

C++11 里还有很多能够让类更优雅的新特性，这里我从“投入产出比”的角度出发，挑出了三个我最喜欢的特性，给你介绍一下，让你不用花太多力气就能很好地改善代码质量。

第一个是“**委托构造**”（delegating constructor）。

如果你的类有多个不同形式的构造函数，为了初始化成员肯定会有大量的重复代码。为了避免重复，常见的做法是把公共的部分提取出来，放到一个 `init()` 函数里，然后构造函数再去调用。这种方法虽然可行，但**效率和可读性较差**，毕竟 `init()` 不是真正的构造函数。

在 C++11 里，你就可以使用“委托构造”的新特性，一个构造函数直接调用另一个构造函数，把构造工作“委托”出去，既简单又高效。

 复制代码

```
1 class DemoDelegating final
2 {
3 private:
4     int a;                // 成员变量
5 public:
6     DemoDelegating(int x) : a(x)    // 基本的构造函数
7     {}
8
9     DemoDelegating() :            // 无参数的构造函数
10        DemoDelegating(0)        // 给出默认值，委托给第一个构造函数
11    {}
12
13    DemoDelegating(const string& s) : // 字符串参数构造函数
14        DemoDelegating(stoi(s))    // 转换成整数，再委托给第一个构造函数
15    {}
16};
```

第二个是“**成员变量初始化**”（In-class member initializer）。

如果你的类有很多成员变量，那么在写构造函数的时候就比较麻烦，必须写出一长串的名字来逐个初始化，不仅不美观，更危险的是，容易“手抖”，遗漏成员，造成未初始化的隐患。

而在 C++11 里，你可以在类里声明变量的同时给它赋值，实现初始化，这样**不但简单清晰，也消除了隐患**。

 复制代码

```
1 class DemoInit final                                // 有很多成员变量的类
2 {
3 private:
4     int          a = 0;                            // 整数成员，赋值初始化
5     string       s = "hello";                      // 字符串成员，赋值初始化
6     vector<int>   v{1, 2, 3};                      // 容器成员，使用花括号的初始化列表
7 public:
8     DemoInit() = default;                          // 默认构造函数
9     ~DemoInit() = default;                         // 默认析构函数
10 public:
11     DemoInit(int x) : a(x) {}                     // 可以单独初始化成员，其他用默认值
12 };
13
```

第三个是“**类型别名**”（Type Alias）。

C++11 扩展了关键字 `using` 的用法，增加了 `typedef` 的能力，可以定义类型别名。它的格式与 `typedef` 正好相反，别名在左边，原名在右边，是标准的赋值形式，所以易写易读。

 复制代码

```
1 using uint_t = unsigned int;                       // using别名
2 typedef unsigned int uint_t;                       // 等价的typedef
3
```

在写类的时候，我们经常会用到很多外部类型，比如标准库里的 `string`、`vector`，还有其他的第三方库和自定义类型。这些名字通常都很长（特别是带上名字空间、模板参数），书写起来很不方便，这个时候我们就可以在类里面用 `using` 给它们起别名，不仅简化了名字，同时还能增强可读性。

```
1 class DemoClass final
2 {
3 public:
4     using this_type      = DemoClass;           // 给自己也起个别名
5     using kafka_conf_type = KafkaConfig;        // 外部类起别名
6
7 public:
8     using string_type    = std::string;         // 字符串类型别名
9     using uint32_type    = uint32_t;           // 整数类型别名
10
11     using set_type       = std::set<int>;        // 集合类型别名
12     using vector_type    = std::vector<std::string>; // 容器类型别名
13
14 private:
15     string_type    m_name = "tom";             // 使用类型别名声明变量
16     uint32_type    m_age  = 23;                // 使用类型别名声明变量
17     set_type       m_books;                    // 使用类型别名声明变量
18
19 private:
20     kafka_conf_type m_conf;                    // 使用类型别名声明变量
21 };
```

类型别名不仅能够让代码规范整齐，而且因为引入了这个“语法层面的宏定义”，将来在维护时还可以随意改换成其他的类型。比如，把字符串改成 `string_view`（C++17 里的字符串只读视图），把集合类型改成 `unordered_set`，只要变动别名定义就行了，原代码不需要做任何改动。

小结

今天我们谈了“面向对象编程”，这节课的内容也比较多，我划一下重点。

1. “面向对象编程”是一种设计思想，要点是“抽象”和“封装”，“继承”“多态”是衍生出的特性，不完全符合现实世界。
2. 在 C++ 里应当少用继承和虚函数，降低对象的成本，绕过那些难懂易错的陷阱。
3. 使用特殊标识符“final”可以禁止类被继承，简化类的层次关系。
4. 类有六大基本函数，对于重要的构造 / 析构函数，可以使用“= default”来显式要求编译器使用默认实现。
5. “委托构造”和“成员变量初始化”特性可以让创建对象的工作更加轻松。
6. 使用 `using` 或 `typedef` 可以为类型起别名，既能够简化代码，还能够适应将来的变化。

所谓“仁者见仁智者见智”，今天我讲的也只能算是我自己的经验、体会。到底要怎么用，你还是要看自己的实际情况，千万不要完全照搬。

课下作业

这次的课下作业时间，我给你留两个思考题：

1. 你对“面向对象编程”有哪些认识，是否赞同这节课的观点？（希望你大胆地说出来，如果意见有分歧，那也很正常，我们一起讨论。）
2. 你觉得应用这节课讲到的准则和技巧能否写出一个“好”的类，还缺什么吗？

欢迎你在留言区写下你的思考和答案，如果觉得今天的内容对你有所帮助，也欢迎分享给你的朋友。我们下节课见。

课外小贴士

1. 要想从理论高度上学好面向对象编程，必须要掌握的知识是“设计模式”，然后还有“开闭原则”“里氏替换原则”等基本原则。
2. 在C++里，不要再使用“`typedef struct {...} xxx;`”的方式来定义结构体，这是传统的C做法，在C++里不仅没有必要，而且会造成困惑。
3. C++11的新特性“move”可以消除对象拷贝的成本，现有的资料大多数把它翻译成“移动”，但我却喜欢用“转移”这个词，更贴切形象、好理解。
4. C++11还提供一个特殊标识符“override”，显式地标记虚函数重载。在我的印象里，好像在20世纪90年代初，C++就有这个关键字，可能是后来C++98标准化时给去掉了。
5. 传统的类编写方式是“*.h+*.cpp”，声明与实现分离，但我更推荐在一个“*.hpp”里实现类的全部功能，这样更“现代”。很多开源的现代C++项目都全面采用了“hpp”的方式，比如著名的Boost。

6月-7月课表抢先看

充 ¥500 得 ¥580

赠「¥ 118 月球主题 AR 笔记本」



【点击】图片, 立即查看>>>

© 版权归极客邦科技所有, 未经许可不得传播售卖。页面已增加防盗追踪, 如有侵权极客邦将依法追究其法律责任。

上一篇 04 | 编译阶段能做什么: 属性和静态断言

下一篇 06 | auto/decltype: 为什么要有自动类型推导?

精选留言 (37)

写留言



Jason

2020-05-16

老师讲的很实用, 读完很有收获, 赞。小贴士里面提到的5, 耳目一新, 确实很有道理, 其他语言如Java、C#、Python的源码文件都是一种类型, 只有咱们c++是头文件和实现文件。曾经有Java同事跟我闲聊, 你们C++这个头文件啊, 鸡肋, 我楞了一下, 思考了几秒钟, 竟然没有反驳的理由。

展开

作者回复: 因为C++继承C, 而C这么做是有历史原因的, 当时的计算机性能弱内存小, 头和实现分离才方便处理。

现在的计算机性能大幅度提升, 所以这种方式也就没有太多必要了, 我建议尽量用hpp的方式, 和其他语言保持一致。

4

9



lckfa李钊

2020-05-16

关于老师的思考题，我个人的想法是本节的知识点还不足以写一个好的类，文中编码准则和常用技巧里的介绍只是在编码层面给出了建议，但是缺少方法论。少用继承，多用组合，这个建议可以再扩展深入点，比如有的鸟不会飞的例子，其实可以将Fly从Bird类抽离出来，将Fly实现成一个独立的接口类，和Bird类进行组合。

另外既然面向对象的核心是 抽象 和封装，封装可以不言自明，但是抽象是个啥，没有言...
展开

作者回复:

1.我的理解，抽象就是“建模”，用类在代码里建立一个现实对象的映射。

2.设计模式更强调对象组合，而不是继承，类模式很少。

3.继承是一种“硬”代码复用，关系比较强，不如组合灵活，我建议少用。当然这还是要建立在对继承等特性有比较深刻认识的基础上。

1

6



软件赚硬幣

2020-05-16

罗老师，我看标准库和boost库很多继承都超过3层哦。比如iostream里面的，ios_base到os，再到istream，再到iostream（同时继承ostream），再到fstream。用了三四层继承和虚继承（多重继承）

作者回复: 现实中有很多深层次继承的例子，但不是说这就是好的，实际上iostream就被很多人批评过。

5

5



嵇斌

2020-05-16

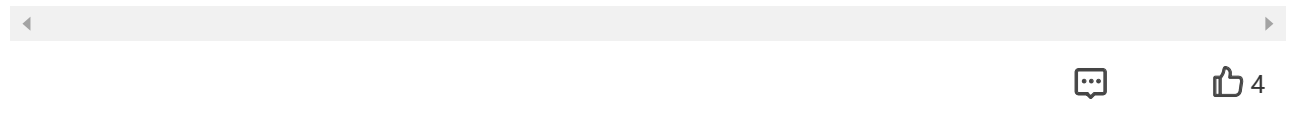
1. 面向对象的首要原则 SRP，单一职责原则。这一点特别赞同。另外就是慎用继承，尽量使用组合去实现。分享一个小故事，之前因为代码要写单元测试，使用到了Google Mock，一开始以为只有虚类才没被Mock，导致很多完全没有必要使用virtual的类使用了virtual，直到有一天看了文档：<https://github.com/google/googletest/blob/master/googl>

emock/docs/cook book.md#mocking-non-virtual-methods-mockingnonvirtualm...

展开 ▾

作者回复: 现在C++有很多工具, 比如智能指针, 可以在外部帮着管理资源, 还有对象池模式, 集中申请释放。

所以RAII还是要看情况, 当然, 如果资源确实是与类强相关, 那么就用RAII自己管理。

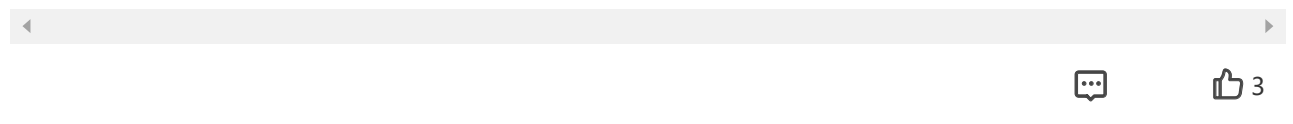


Eglinux

2020-05-16

在 .h 中将类的定义和实现写在一起, 这样不是默认所有成员函数都内联了吗?

作者回复: 是的, 但是否内联是由编译器决定的, 通常只有小的函数才会内联, 大的函数不会内联, 因为反而成本高不划算。



被讨厌的勇气

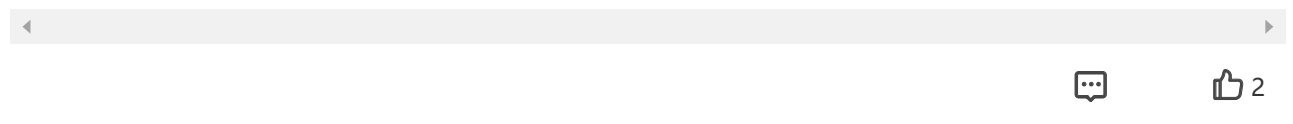
2020-05-17

将声明与实现放在同一个.hpp文件中, 会不会降低编译速度? 仅include声明, 和include声明和实现对编译速度有什么影响?

展开 ▾

作者回复: 我觉得在现在主流的cpu上, 差异应该不大。可以看一下标准库, 全是头文件, 可见标准委员会对这个也是有信心的。

如果还是不放心, 可以做一下测试, 但可能只有超大项目才能测试出差异来。



Raysuner

2020-05-23

本菜鸟表示老师讲的很好, 但我还是没有什么具体的思路能写出好的类。老师的方法经过提炼总结, 对经验丰富的人员应该比较有帮助, 但我还比较缺乏经验, 显得有些笼统, 希望老师以后能谈谈到底怎么用这个术, 什么情况下用这个术

展开 ▾

作者回复: 学习C++, 还有任何编程语言, 都要有适当的实践, 不然就只是“花架子”。

可以看一些开源项目, 从中领会这些设计原则。



1



MrTin

2020-05-21

另外想请教老师, 想转行做C++的话, 主要可以做哪些方面事情呢?

作者回复: 目前C++主要做系统编程, 比较偏底层, 不像做应用那么范围广。可以去招聘网站上看看, C++职位都有哪些, 再决定是否转。



1



robonix

2020-05-21

定义一个新的名字空间, 把内部类都“提”到外面, 降低原来类的耦合度和复杂度。老师, 这句话没看懂, 能加一个简单的代码示例吗?

展开 ▾

作者回复: 大概就是这样

~~~

```
namespace xxx {
```

```
    class inner_class {...};
```

```
    class big_class {...};
```

```
}
```

~~~



1



java2c++

2020-05-18

老师好, 这一点我不太赞同“使用 using 或 typedef 可以为类型起别名, 既能够简化代码, 还能够适应将来的变化。”类型别名看似简化代码, 实际上增加了阅读代码的难度, 最近看公司c++老系统里一个系统里同一个类搞出好几个别名, 太影响阅读效率了

展开 ∨

作者回复: 这个要看怎么用了。

我说的是在类内部定义别名，控制别名的作用域，如果在外部定义别名就是另外一回事了。

◀ ▶



1



亮

2020-05-16

课外小贴士的2应该怎么来定义结构体更好?

展开 ∨

作者回复: 直接用struct，和class一样。

我的是意思是不要再用C的typedef方法来定义结构体的别名了，没有必要。

◀ ▶



2



Confidant.

2020-05-16

我正在学习设计模式里面的思想，刚觉得virtual帮我们设计类省了很多事情，纯虚基类还是很有必要的，这样可以在二进制层面复用代码，把链接推迟到运行时期，避免了我们在修改代码的时候，牵一发而动全身

作者回复: 用virtual就要用跟它相关的一大堆特性了，属于比较“纯”的面向对象。

但在现代C++里，可以用泛型来达到类似的效果，而且性能更高。

◀ ▶



2



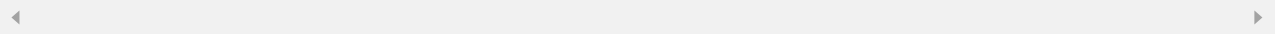
范闲

2020-05-16

- 1.cpp中的面向对象是建立在封装，继承和多态上。其中继承和动态多态强相关。很多情况下类的继承应该是is-a，has-a的算是类的组合。
2. 好的类个人认为应该可以表意。从设计上需要考虑类的几个构造是否都需要，从继承上考虑是否有作为基类的可能。

展开 ∨

作者回复: 说的很好。



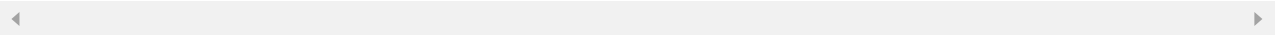
阿鼎

2020-05-16

我的体会，设计模式，是为了应对变化，大量使用继承，绕来绕去，不如直接把可能变化的地方，用function&bind来的直接。

展开 ∨

作者回复: 对，有的时候用function会比虚函数、抽象类更灵活，这就是对象组合，也是策略模式。



TC128

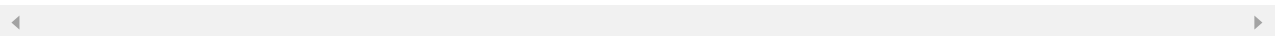
2020-05-28

老师，请教个问题。前几天刚刚学习了设计模式，里面有个“接口倒置原则”要用到虚函数，那不用虚函数这个原则这实现呢？

展开 ∨

作者回复: 像适配器、桥接这样的模式，可以使用泛型（鸭子类型）来组合对象，也可以用std::function来存储调用的函数、lambda表达式。

我觉得要根据实际情况去设计、挑选工具，现在泛型还没有那种比较成熟，能拿来就用的模式。



Geek_54b85f

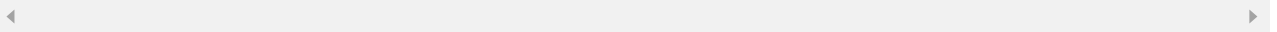
2020-05-28

之前是从写java临时转到C++开发的，面向对象和抽象类的概念一直深入人心，导致项目里大量用了纯虚类和虚函数，现在也觉得过于冗余厚重，老师能否将讲下如何用模版和组合的方式改造这种陈旧的设计呢？谢谢！

作者回复: 这个话题比较大，一下子说不清。

我觉得，首先要深刻理解设计模式，改继承为组合。然后，就可以利用泛型/模板了，把类型“参数化”。

可惜的是关于泛型编程没有经典书籍和指导原则，只能自己在实际中多摸索，看标准库和好的开源项目是一个比较不错的办法，借鉴它们的成功经验。



钟健鑫

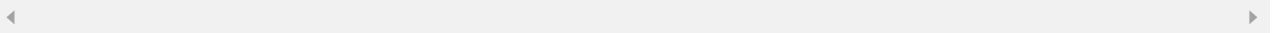
2020-05-26

“面向对象编程”是一种设计思想，要点是“抽象”和“封装”，“继承”“多态”是衍生出的特性，不完全符合现实世界。

这一点不认同，抽象和封装 在面向对象范式中恰恰不是核心，因为C和其它语言都能做到，而对于OO来说 通过多态实现泛化，让系统能支持扩展点处的易扩充 才是核心
展开 ∨

作者回复: 这点确实有很多争议，大家多讨论才能认识的更清楚。

你说的有一点很对，C和其他语言也能实现抽象和封装。但这个也是面向对象，它并不是限于某种语言的，并不是说面向过程的语言就不能应用面向对象的范式，所以我觉得这才是面向对象的核心。



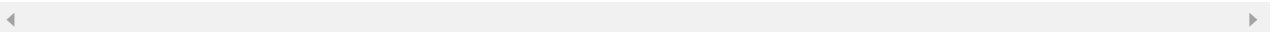
Seven

2020-05-25

这节课我喜欢 😊

展开 ∨

作者回复: thanks，后面的课程也要努力。



Eason Tai

2020-05-24

有个小问题，为什么要在一些类里面多次书写 public 和 private 关键字呢？是增加可读性或者强调什么嘛？

作者回复: 是的，这大概是借鉴了一点java吧，用public、private来分组不同的逻辑段落，增强可读性。



sugar

2020-05-22

老师讲的都是干货，先赞一个。这些东西能感觉到是老师的实际经验得来，我在书上是学不到的。想问下，老师能否也谈谈其他编程语言里在oop方面推出的interface，这个东西似乎比class继承要轻和低耦，golang甚至直接干掉了继承 只允许借口实现，老师能谈谈看法吗

展开 ∨

作者回复: 实话实说，这些都是C++ “玩剩下的”。

原因就是多重继承还有普通的继承太重，所以用接口继承就很轻。

接口继承算是面向对象的典型用法了，在Java、Go这样的“纯”面向对象语言里也是唯一的解法（很多年前的认识，现在也许不一样），但C++还可以用泛型，所以更灵活。

喜欢Java、Go在C++里这么用当然没问题，但也应该再看看泛型编程、函数式编程，可能会“柳暗花明”。

